На данный момент мы с вами в целом познакомились со способами создания объектов различных классов. При этом, как бы мы их не создавали, все они формируются по единой схеме. Что это за схема? Давайте представим, что у нас простой класс Point:

```language-cpp
class Point {
    int x {-1};
    int y {-1};

public:
    Point() : x(0), y(0)
        { }
    Point(int a, int b) : x(a), y(b)
        { }

    void get_coords(int& a, int& b) { a = x; b = y; }
    void set_coords(int a, int b) { x = a; y = b; }
};
```

Его объекты можно сформировать следующими командами:

```language-cpp
int main()
{
    Point pt1, pt2(1, 2);
    Point* ptr_pt1 = new Point;
    Point* ptr_pt2 = new Point(10, 20);

    delete ptr_pt1;
    delete ptr_pt2;

    return 0;
}
```

Что же стоит за словами «создание объекта» и его «уничтожение»? В действительности, все происходит по следующим этапам:

1. Выделение памяти под объект.
2. Инициализация переменных объекта.
3. Вызов конструктора объекта.
4. Жизнь объекта.
5. Вызов деструктора объекта.

6. Удаление переменных объекта (вызов их деструкторов, при необходимости).
7. Освобождение памяти, занимаемой объектом.

И так для любых объектов класса, как бы мы их ни создавали (разумеется, речь идет о корректных способах).

Обратите внимание, что после выделения памяти происходит инициализация переменных и только после этого вызов конструктора. В частности, это означает, что инициализация переменных, прописанная в классе:

```language-cpp
class Point {
    int x {-1};
    int y {-1};
...
};
```

отрабатывает до вызова конструктора. То же самое касается и списка инициализации у конструкторов:

 ```cpp
Point() : x(0), y(0)
{ }
Point(int a, int b) : x(a), y(b)
{ }
```

```
Переменные x, y инициализируются нулями или значениями a, b, причем, эта инициализация имеет более высокий приоритет перед инициализацией, записанной в классе. То есть, при создании объекта, например, командой:

```cpp
Point pt;
```

переменные x, y будут инициализированы нулями, а не -1. Инициализация с -1 будет просто проигнорирована.

Также обратите внимание, что инициализация, записанная в классе, – это лишь объявление инициализации, а не реализация. Отрабатывает она только при создании объекта класса в самом объекте, т.к. только в нем появляются сами переменные x, y. До этого они не существуют, а значит, и не могут быть инициализированы.
```

Следующий важный факт, вытекающий из схемы жизни объекта, связан с использованием объектов одних классов внутри другого класса. Например, перед классом Point объявим еще один класс:

```cpp
class Log {
public:
    Log()
        { }
};
```

А в классе Point сделаем объявление объекта класса Log:

```cpp
class Point {
    int x {-1};
    int y {-1};
    Log lg;
public:
    Point() : x(0), y(0)
        { }
    Point(int a, int b) : x(a), y(b)
        { }
...
};
```

Спрашивается, в каком порядке будут вызваны конструкторы этих классов? Схема жизни объекта дает нам однозначный ответ: **сначала в блоке инициализации будет создан объект lg и, соответственно, вызовется конструктор класса Log, а затем, вызывается конструктор класса Point**. То есть, конструктор класса Point вызывается только после формирования всех переменных и объектов, описанных в этом классе.

Мало того, если в классе Log не будет конструктора по умолчанию, например:

```cpp
class Log {
    unsigned id {0};

public:
    Log(unsigned id_log)
        { id = id_log; }
};
```

То при создании объектов класса Point возникнет ошибка невозможности формирования объекта lg. В этом случае нам нужно явно передать один

аргумент при создании объекта lg. Сделать это можно либо непосредственно в классе:

```language-cpp
class Point {
    int x {-1};
    int y {-1};
    Log lg {5};
...
};
```

Либо в списке инициализации соответствующих конструкторов:

```language-cpp
class Point {
    int x {-1};
    int y {-1};
    Log lg;

public:
    Point() : x(0), y(0), lg(1)
        { }
    Point(int a, int b, unsigned id=1) : x(a), y(b), lg(id)
        { }
...
};
```

Так как список инициализации имеет более высокий приоритет, то в объекте переменная lg будет формироваться с передачей одного аргумента и вызова конструктора с одним параметром.

Соответственно, деструкторы вызываются в обратном порядке: **сначала деструктор для объекта текущего класса, а затем, деструкторы объектов переменных**. В конце происходит освобождение памяти, занимаемой объектом. Правда, не всегда этот процесс выполняется автоматически. Следует помнить, что если объект был создан с помощью оператора new:

```language-cpp
Point* ptr_pt = new Point;
```

то в конце мы самостоятельно должны сделать освобождение:

```language-cpp
delete ptr_pt;
```

Иначе, память останется выделенной, но, скорее всего, не используемой.

Вот такие этапы формирования и уничтожения объектов классов следует знать для их грамотного использования в программах.